Multi-agent System Simulation in Scala: An Evaluation of Actors for Parallel Simulation

Aaron B. Todd[1], **Amara K. Keller**[2], **Mark C. Lewis**[2] **and Martin G. Kelly**[3]
[1]Department of Computer Science, Grinnell College, Grinnell, IA, USA
[2]Department of Computer Science, Trinity University, San Antonio, TX, USA
[3]Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

Abstract— *Multi-agent system (MAS) simulation, a growing field within artificial intelligence, requires the creation of high-performance, parallel, and user-friendly simulation frameworks. The standard approach is to use threads and shared memory. The drawbacks of this approach are the common concurrency pitfalls of race conditions and performance loss due to synchronization. Our goal was to evaluate the feasibility of an alternate model of concurrency, actors. An actor can be thought of as a very lightweight thread that does not share memory with other threads, instead communicating only through message passing. Actors seem to be a natural fit for this task, since agents are concurrently processed objects that communicate with each other through message passing. We write an actor framework and an equivalent threaded framework in the modern object-functional JVM language Scala and compare their performance. We conclude that the actor model seems like a natural fit, but its performance is inferior to that of the threaded model. Despite this drawback, it shows great promise due to its elegance and simplicity. When scaling to multiple machines, the advantages of actors will almost certainly outweigh any performance costs.*

Keywords: MAS, Scala, Parallel Simulation, Actors, AI

1. Introduction and Background

An important simulation problem is that of multi-agent systems (MAS). A MAS "can be defined as a loosely coupled network of problem solvers that interact to solve problems that are beyond the individual capabilities or knowledge of each problem solver. These problem solvers, often called agents, are autonomous and can be heterogeneous in nature" [1]. Because the definition of an agent is so broad, MAS can model many different situations, such as economic, social, and political activity. Of course, creating realistic simulations is not easy and requires collaboration with economics and/or other social sciences. However, MAS provide the technological framework to make such modeling possible.

Implementations of MAS frameworks vary widely, but the core parallelization model behind many of them is that of threads and shared memory. Although this approach can work, it suffers from the hazards of concurrent data modification and other race conditions. These issues make testing and debugging of the frameworks very difficult, and solutions to these problems typically incur performance penalties and programmer headaches.

Another approach to parallel programming is the actor model [2]. An actor can be defined as a lightweight process that communicates with other actors through message passing. These messages are buffered in the actors' mailboxes for the actor to respond to. Actors do not share any memory with other actors, and they all process concurrently. Message passing is asynchronous. This model avoids the numerous shared memory pitfalls associated with the conventional threaded model of concurrency. The best known actor implementation is in the language Erlang, a functional language designed for efficient fault-tolerant distributed systems [3].

A MAS framework based on this actor model would avoid many of the concurrency issues afflicting the threaded frameworks. In such a framework, it would be natural for each agent to be an actor. Communication between agents would then be done by the actor's message passing methods and the simulation would be implicitly parallel. While it would help eliminate many reliability issues and make frameworks much easier to write, this approach is only practical if the performance is comparable to that of the threaded frameworks. Our goal is to perform a performance evaluation of an actor model MAS framework written in the language Scala by comparing it to a comparable threaded framework.

Scala is a fairly new programming language that has been in development at EPFL in Switzerland since 2003. It uses an object-functional paradigm and compiles to the JVM, which allows seamless calls to Java libraries and code. It has static type checking and, as a result, takes full advantage of HotSpot JVM implementations and typically runs at the same speed as Java programs [4].

There has been significant work on the Scala language that is well documented in the field of programming languages. As a result, the language pulls in many of the best ideas from the field [5] [6] [7] [8]. The name Scala is short for Scalable Language, a property that makes it ideal for generating Domain Specific Languages (DSLs). This can be extremely beneficial in the field of simulation, where many simulation packages have basically built up their own

DSLs over the years [9]. The design of Scala allows libraries to be written such that they look and operate as normal language features. These features, among many others, were significant factors in motivating us to use the language for these frameworks. The Scala actor library is unusually high-performing relative to other JVM actor implementations. Together, Scala's extensibility and the convenience of a JVM platform makes Scala a great language choice [10]. This is the reason why we chose it over other actor-supporting languages such as Erlang.

2. Related Work

To our knowledge, we are one of the first groups to consider using Scala's actor library for parallel simulations. Some research has been done into the feasibility of using Erlang for MAS. The implementation described by Varela et al. does map Erlang's lightweight processes directly to agents, but in this case, each agent is a collection of these processes. Unfortunately, Varela's work does not provide a performance evaluation [11]. His group's primary motivations for using Erlang over Java was the much more natural fit they saw between Erlang and the coding of agent behavior, combined with Erlang's excellent support for distributed computing [**?**]. Performance appears to have been good enough that any weakness was compensated for by the convenience of using a language that fits the problem well.

3. Scala

Scala, as a language very closely related to Java, borrows much of its syntax; however, it omits semicolons and eliminates the need for extensive boilerplate code. The most noticeable difference is a type inference system for limiting type specification when such specification would be redundant. These tweaks make Scala much easier to read and feel more like a scripting language even though it actually runs on the robust JVM platform.

Listing 1: Scala Int to String class.

```
class Foo {
  def bar(arg: Int): String = arg.toString
}
```

Listing 2: Java Int to String class.

```
public class Foo {
  public String bar(Int arg) {
    return Integer.toString(arg);
  }
}
```

In addition to these simple variations, Scala has many more subtle differences and features, including very natural

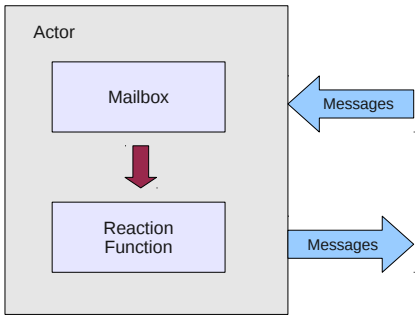

Fig. 1: An actor. Mailbox buffers messages until the reaction function processes them.

support for pattern matching. This is very useful because the primary action of an actor is to react to a message with an action that varies based on message content. Excellent pattern matching syntax makes this a painless process.

Scala can represent this reaction function as a number of case statements. These are matching functions evaluated in sequence on the input. These can match on type or value and can pull inner values by automatically applying extractors. Extractors are present in all standard library collections and can be automatically generated for a class by declaring it with the case keyword. In the following example, the input message is checked to see if it is an Int, the String "foo", or a Bar that contains 5. A default is provided, but since only one match can occur, it does not require special syntax.

Listing 3: A simple match block.

```
msg match {
  case msg: Int => println(``An integer.'')
  case ``foo'' => println(``Found foo.'')
  case Bar(5) => println(``The bar.'')
  case _ =>
}
```

4. Actor Framework

4.1 Scala Actors

Scala has an excellent actor library. Since Scala is a JVM language, this library is quite complicated because the JVM is not designed to run actors efficiently [12] [13]. The library works around this limitation by leveraging the language's functional nature and by using exceptions to navigate the call stack. Event based actors, a variant that does not block an underlying thread while waiting for messages, are implemented as closures waiting to be called with an input message. Upon receiving a message, the actor's associated closure is then scheduled on an available executor thread. When an actor is finished reacting to the message, it throws a suspend actor exception, which returns the actor to its idle

state. By implementing actors in this way instead of mapping them directly to JVM threads, they become very lightweight. This gives them substantially better performance than many other JVM-based actor implementations [10].

A Scala actor can be defined by extending the Actor trait. All actor functionality is contained in the act method, which must be defined by the subclass. A simple actor is defined below. React is the method actors call to "react" to the next message in their mailbox. Due to its non-returning nature, a result of its exception-based implementation, a for or while loop will not work, so the library provides a special loop function. Inside react is a matching function similar to the one above. The "!" method is used to send messages to actors, and sender can be used to refer to the actor that sent the message being reacted to. This actor responds to Pings with Pongs and Pongs with Pings.

Listing 4: A Ping-Pong actor.

```
class PingPong extends Actor {
  def act() {
    loop {
      react {
        case Ping => sender ! Pong
        case Pong => sender ! Ping
      }
    }
  }
}
```

4.2 Agents

In our framework, each agent is a subclass of actor. This design allows the framework to inherit all of the actor message passing functionality, and since actors process concurrently, there are only a few agent features left to implement. The primary task is to convert the event based system of actors reacting to a system in which agents iterate through time steps. We do this by creating a clock actor that sends agents messages which trigger the processing of steps. Once each agent finishes its step, it sends a message back to the clock indicating that it is finished. Once the clock receives finished messages from each agent, it sends out new DoStep messages. The method that agents call upon receiving a DoStep message is called doStep.

4.3 Stepping Algorithm

This results in the following step algorithm for the framework. A runner object initializes the simulation and then the clock object loops a step function of the form:

- Send DoStep message to each agent.
- Wait until every agent has replied with an EndStep message.
- Cleanup and repeat if not finished.

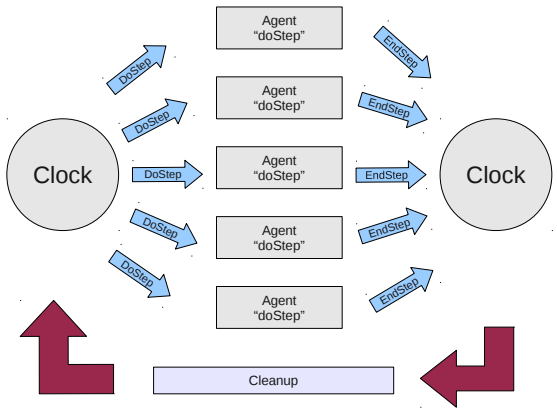

Fig. 2: Clock sends DoStep messages, Agents call doStep method, Agents inform clock that they are finished. Cleanup and repeat.

Inside the doStep call, agents perform any simulation logic they need to with the option of placing some code to handle messages in a dedicated handleMessage method. This was done because many messages in the simulation are simple information requests. Processing these requests inside doStep creates an unnecessary hassle because they interrupt the logical flow of the agent's code. This external method also allows agents to respond to information requests once finished with their core logic without having to know how many requests they will receive. In the case in which receiving a message is a core part of the agent's logic, they can still react to it in doStep.

These methods are defined in a core Agent abstract class that is extended by simulation writers as needed. The act method of an agent is a simple loop reacting to messages. The messages it receives are of two possible types. The first is FrameworkMessage. This type of message is used by the framework to tell the agent what it should be doing. Examples include messages to start the simulation and do steps. The second type is a SimulationMessage. This message is sent between agents as part of their steps. An example is an information request message. The react sends each type to its appropriate handleMessage method. In the case of a SimulationMessage, this is the handleMessage method defined by the simulation writer.

Upon receiving a DoStep message, the agent calls the doStep method written by the simulation writer. Inside this method, agents are free to send messages to other agents using the conventional agent message send method, "!". When reacting to messages, a special react method specifically for agents must be used. This implementation provides a number of benefits. Using a number of Scala language features allows this method to return normally, unlike the usual react. It also allows an agent to use its

handleMessage method in addition to the partial function passed in to react to messages and does so in a way that is transparent to the simulation writer.

Listing 5: An example agent subclass.

```
class AnAgent(friend: AnAgent) extends Agent {
  def doStep() {
    friend ! Greet(``Hello'')
    agentReact {
      case Greet(msg) => println(msg)
    }
  }
}
```

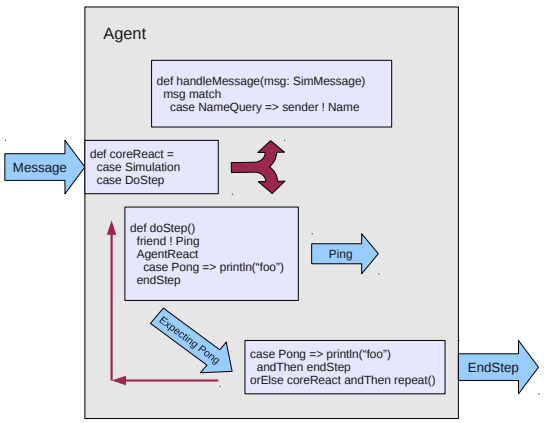

Fig. 3: Agent message processing. If SimulationMessage, send to handleMessage. If DoStep, perform agent behavior. If agentReact called, move to modified react which forwards message back to coreReact and then repeats agentReact by calling the saved continuation if not the expected message.

4.4 Implementation

The implementation of this react is the most complex component of the actor framework. While it is not strictly necessary, its inclusion makes writing simulations substantially easier through the avoidance of bizarre control flow. For that reason, the performance hit from its overhead was deemed acceptable.

The actual implementation is as follows: When agentReact is called, the first action taken is to use Scala's delimited continuations library to store all remaining computation in the doStep method as a continuation. Then the agent reacts to incoming messages with both the partial function given to agentReact and the agent's coreReact partial function. If the supplied partial function is used to react to the message, the continuation is then called using the actor library method andThen, which takes a function to apply once react has finished. If the message is handled by the coreReact partial function, this react block is repeated, again by calling andThen. The primary drawback to this system is that all looping constructs with continuation-creating code in their bodies must be implemented with an understanding of continuations. Since Scala's constructs are not aware, a special agentReactWhile function is defined.

Scala's continuations library is one of the few that is not implemented purely as library code. It works by using a compiler plug-in to perform a transformation of all code contained in the delimited continuation to continuation-passing style (CPS). The result of this transformation is that instead of returning normally, all functions take an argument, which is the function to apply to the result. This function can then be saved and stored as a "continuation" instead of being evaluated immediately. The details of this transformation are available in [14].

One might question the use of the Scala continuations library to achieve this behavior instead of using andThen following react. While the latter probably has better performance, it would result in fairly convoluted code. If an agent had many reacts interspersed with pieces of computation, using andThen would require a deep nesting of function

Fig. 4: Note how a normal react results in deep nesting.

(a) Returning react.

```
def doStep() {
  agentReact {
    partialFunction
  }
  agentCode
  agentReact {
    partialFunction
  }
  agentCode
}
```

(b) Normal react.

```
def doStep() {
  agentReact {
    partialFunction
  } andThen {
    agentCode
    agentReact {
      partialFunction
    } andThen {
      agentCode
    }
  }
}
```

calls. When writing more complex logic, this nesting would begin to make code unreadable. One way to think of our use of continuations is that the compiler's CPS transform is simply a way to get rid of these extra brackets in order to make the code more readable. In the original work on Scala actors, the authors also state that the lack of a return on react is due to the lack of first-class continuations and, had there been a technique similar to our implementation, they would have used it instead [12]. When continuations were added to Scala, one of the examples used was a modification to react that made it return, which was an implementation very similar to ours [14].

Messages in this framework are defined very simply. FrameworkMessage and SimulationMessage are both traits that extend a Message trait. Framework messages with parameters are defined as case classes, where case is a Scala keyword that automatically generates simple constructor methods and extractors for the pattern matching done

in react. If there are no parameters, singleton messages are used to save memory. Simulation writers define their message types by extending the SimulationMessage trait with additional case classes.

There is also a CentralAgent class that is given information about all agents in the simulation. This agent is useful because it allows normal agents to perform actions such as requesting a reference to a random agent very easily. It is always in a react loop waiting to respond to messages from agents. If they desire, simulation writers can easily extend it with additional functionality.

The final component of the simulation is the system by which agents are initially created. A simulation writer must provide an iterator that produces agents in sequence. Any initial setup, such as setting agent data, must be done by this iterator. A future task is to implement a DSL that simplifies this process.

4.5 Example Simulation

Our primary benchmark is a simulation of CommunicatingAgents. These agents are friends with every other agent, and each step they send a Hello message to each friend. Once they have received a response HelloBack message they finish their step. When doStep is called, the agent sends Hello to each friend and counts how many were sent. It then moves into an agentReact loop, in which it waits for a response from every friend. Concurrently with agentReact, handleMessage responds to Hello messages.

Listing 6: CommunicatingAgent implementation.

```
class CommunicatingAgent
  (val friends: ListBuffer[CommunicatingAgent])
    extends Agent {
  def doStep() {
    var count = 0
    for( a <- friends) {
      a ! Hello
      count += 1
    }
    agentReactWhile(count > 0) {
      case HelloBack => count -= 1
    }
    endStep
  }
  def handleMessage(msg: SimulationMessage) {
    msg match {
      case Hello => sender ! HelloBack
    }
  }
}
```

5. Threaded Framework

In order to evaluate the performance of the actor parallelism approach to MAS, we also wrote simulations in a framework using a threaded model. This framework provides an equivalent environment to write simulations such as ours using conventional concurrency constructs and a master-slave program structure. It was written as an initial exploration of the merits of Scala as a MAS simulation language. The step algorithm is as follows.

- Master calls "runStep" on slave.
- The slave iterates through its agent list.
- "doStep" is called for each agent.
  - Returns a list of messages the agent has sent.
  - Messages to agents placed in their mailboxes.
- Slave processes agents again
  - "handleMessages" is called for each agent.
- Repeats until there are no messages.
- Slave finishes; informs master.
- Cleanup and advance time step.

Unfortunately, this framework suffers from many of the problems associated with the threaded concurrency model. Race conditions between threads make testing and debugging difficult, and frequent synchronization of access to shared memory slowed program execution down.

6. Experiments and Results

6.1 Testing Goals and Methods

Our primary goal in this research was to determine how well Scala's actor library performs relative to threads for MAS simulation. To do this, we run a number of benchmarks for our two frameworks that stress different framework components. For these tests we recorded wall clock time between simulation start and end, not counting agent creation. Our test machines contained dual Xeon 5450 quad-core CPUs at 3.0 GHz and 16 GB of memory. All tests were run with a maximum Java heap size of 14 GB and simulations were run for 20 steps.

6.2 Agent Number

Our first test was to see how execution time scaled while increasing the number of agents. To test this we wrote an agent class where each agent performs no actions and just immediately ends its step after starting. Here we had some very bizarre findings. The threaded framework scaled linearly with the addition of new agents, which is what was expected. The actor framework did not. As Fig. 6 shows, at 100k agents, the execution times become very random. Our plausible explanation for this behavior is the interaction between the actor library, the JVM, and multi-CPU machines. Since actors have special scheduling that leverages exceptions, the clock actor is frequently suspended and resumed. As this is happening, it could be switching between threads, cores, or the physical CPUs. If this were to thrash CPU cache or interfere with exception processing in the JVM, execution would be substantially slowed down

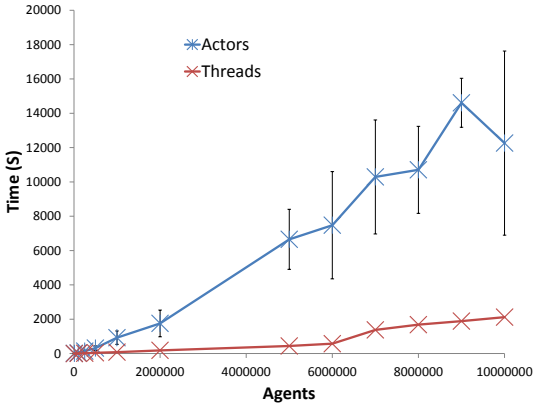

Fig. 5: Execution time scaling for simple agents.

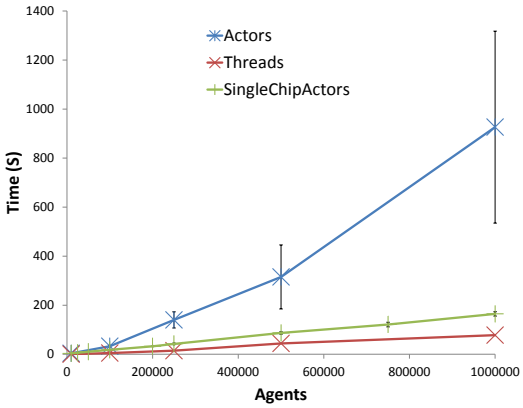

Fig. 6: Note very large uncertainty for dual CPU actors, and essentially none for single CPU actors.

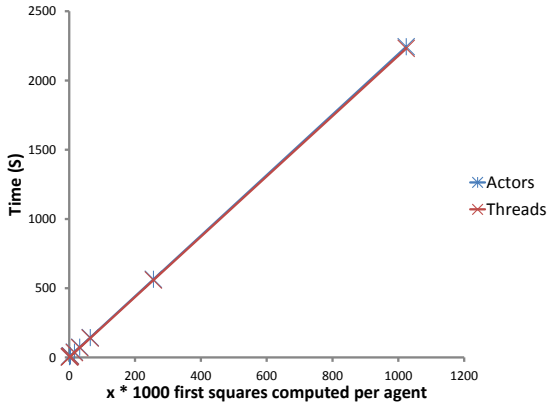

Fig. 7: Execution time scaling as agent computational workload varies.

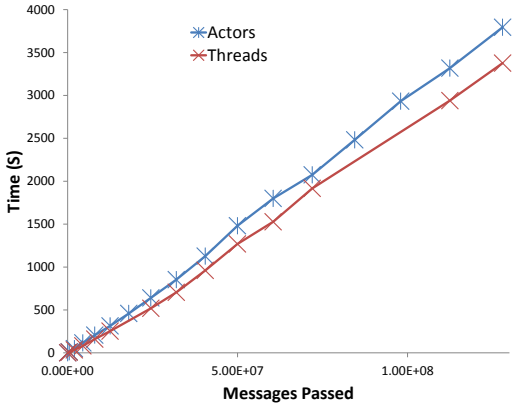

Fig. 8: Execution time scaling as messages passed increases.

by a seemingly random amount. To test this hypothesis, runs were made on a single CPU machine up to 1 million agents (capped due to memory limitations) and no random variation was seen. While this is evidence in support of our hypothesis, the problem merits additional effort to determine the exact causes.

6.3 Computational Workload

The second test was to see how efficiently each framework is for computationally intensive agents. This was done by creating an agent type that computes the first 1000*scaler squares. Our hypothesis was that as agent processor use increases, differences in framework efficiency have less affect on the resulting execution time. This is likely due to the fact that regardless of which framework this agent code is in, it still requires the same amount of CPU time. As Fig. 7 shows, this hypothesis appears to be correct.

6.4 Messaging

Our final test was of message passing performance. This was done using the previously described communicating agents simulation. In it, each agent is friends with every other agent, so messages passed scales quadratically. This was done instead of numerous rounds of messaging for a small constant number of agents because it seems like a more realistic way in which a framework would be stressed. It is also the case that the threaded framework handles many messages in one round very differently from many rounds of a few messages per step.

As seen in Fig. 8, both frameworks exhibit roughly similar linear scaling with messages sent, but the actor framework scales slightly less optimally. This difference can be explained by the fact that starting and ending steps requires message sending in the actor framework.

6.5 Overall Scaling

The overall performance of actors for MAS simulation in Scala was slightly inferior to that of threads. In terms of memory usage, both frameworks were similar, but the threaded framework had measurably better execution time for the messaging benchmark. It also did not suffer any unusual issues on the agent count benchmark. Since the

primary way a MAS will be scaled up is by increasing the number of agents and message volume, from a purely performance perspective, threads are the obvious way to go.

7. Future Work

There are a number of ways in which our actor framework could be improved. One of the main motivations for using Scala was its extensibility, so a DSL for writing agent AI could be implemented. This would make writing agent code much simpler. Another task would be to use Scala's remote actor library to distribute this framework over multiple machines. Remote actors allow local proxy actors to exist for actors on other machines. These proxies can be sent messages that are forwarded to the actual actor. Other approaches to synchronization could also be employed. The current framework requires that all actors compute in lockstep, but this restriction could be loosened as long as agents were required to maintain the ability to respond to requests for information about their past states. A final task could also be to write a stripped down version of the actor library and tune it for the purpose of MAS simulation. This has the potential to eliminate the performance gap between the thread and actor approaches.

8. Conclusion

We set out to compare the relative performance of two approaches to parallelism when applied to the simulation of multi-agent systems in the language Scala. Our conclusion is that the threaded approach is superior from a strictly performance-oriented point of view. However, with the exception of issues relating to the agent number benchmark, actors did perform respectably. Given the ease of writing the actor framework and the numerous ways it could be improved, such as distribution across multiple machines, loosening of time synchronization, or tuning the actor implementation, this model shows clear promise applied to MAS simulation. The language Scala is also an excellent choice due to the ease with which it can be extended to facilitate a DSL for writing agent AI. For these reasons, we recommend that future groups attempting to implement MAS frameworks consider using Scala actors.

References

[1] K. P. Sycara, "Multiagent systems," *AI Magazine*, vol. 19, pp. 79–92, 1998.

[2] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[3] J. Armstrong, "Erlang," *Commun. ACM*, vol. 53, pp. 68–75, September 2010. [Online]. Available: http://doi.acm.org/10.1145/1810891.1810910

[4] "The computer language benchmark game," http://shootout.alioth.debian.org//, 3 2011.

[5] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, "Language virtualization for heterogeneous parallel computing," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 835–847. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869527

[6] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Proceedings of the ninth international conference on Generative programming and component engineering*, ser. GPCE '10. New York, NY, USA: ACM, 2010, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/1868294.1868314

[7] C. Hofer and K. Ostermann, "Modular domain-specific language components in scala," *SIGPLAN Not.*, vol. 46, pp. 83–92, October 2010. [Online]. Available: http://doi.acm.org/10.1145/1942788.1868307

[8] M. Odersky, "The scala experiment: can we provide better language support for component systems?" in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 166–167. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111052

[9] A. Law, *Simulation Modeling and Analysis with Expertfit Software*. McGraw-Hill Science/Engineering/Math, 2006.

[10] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1596655.1596658

[11] C. Varela, C. Abalde, L. Castro, and J. Gulías, "On modelling agent systems with erlang," in *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, ser. ERLANG '04. New York, NY, USA: ACM, 2004, pp. 65–70. [Online]. Available: http://doi.acm.org/10.1145/1022471.1022481

[12] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, pp. 202–220, February 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1496391.1496422

[13] ——, "Event-Based Programming without Inversion of Control," in *Modular Programming Languages*, ser. Lecture Notes in Computer Science, D. E. Lightfoot and C. A. Szyperski, Eds., 2006, pp. 4–22.

[14] T. Rompf, I. Maier, and M. Odersky, "Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform," in *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '09. New York, NY, USA: ACM, 2009, pp. 317–328. [Online]. Available: http://doi.acm.org/10.1145/1596550.1596596